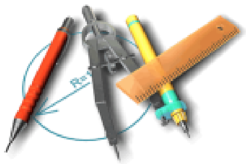
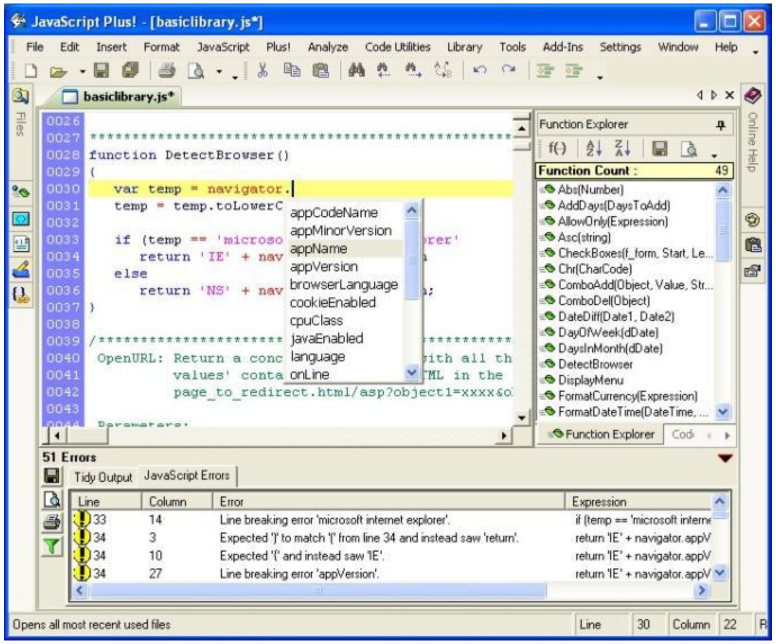




JavaScript tananyag

Könye Attila

A 2. évfolyamos Műszaki informatikus tanulók

„Műszaki programozás gyakorlat” tantárgyához

Széchenyi Középiskola, Zalaegerszeg

Verzió: 2014. augusztus

Bevezetés

JavaScript programozási nyelv egy objektumalapú szkriptnyelv, amelyet weblapokon elterjedten használnak. Eredetileg *Brendan Eich*, a *Netscape Communications* mérnöke fejlesztette ki. Struktúrájában a Java és a C programozási nyelvhez áll közel. A jelenleg érvényes szabvány az **ECMA-262**, ami a JavaScript 1.5-nek felel meg. Ez a szabvány egyben ISO szabvány is.

A **JavaScript kód html** fájlban vagy külön (jellemzően **.js** kiterjesztésű) szövegfájlban van. Ezek a fájlok tetszőleges szövegszerkesztő (nem dokumentumszerkesztő) programmal szerkeszthetőek. A **JavaScript** esetében a futási környezet jellemzően egy webböngésző (JavaScript-motorja). Windows környezetben futtatható a **wscript.exe** segítségével, vagy Linuxos környezetben **nodejs**-el futtatható.

Ez a könyv a *Széchenyi István Szakközépiskola* Műszaki informatikus tanulók „Műszaki programozás gyakorlat" tantárgyának tanmenetét követi. Óráról órára jegyzetet, ismétlési lehetőséget, gyakorlást biztosítva segíti a haladást.

A leckék végén -a **függelék**ben- gyors összefoglalót találunk az utasítások szintaktikájáról, az operátorokról, gyakran alkalmazott objektumokról.

A **források** fejezet további kiegészítőket, tananyagokat, referenciákat tartalmaz.

A dokumentum mérete a képernyős olvasáshoz (és vetítéshez) igazodott. Mielőtt kinyomtatnád, gondolj a környezetre, a papírra, a hódokra, meg a világbékére.

konye@centernet.hu

1. lecke „Hello World”, avagy laza alapozás

- Kedvenc szövegszerkesztőnkben hozzunk létre egy új üres HTML (HTML multihighlighter a javasolt) dokumentumot.
- Mentsük el **.html** kiterjesztéssel.
- Nyissuk meg a fájlt kedvenc böngészőnkben.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8">
  <title>JavaScript tananyag</title>
  </head>
  <body>
  <script type="text/javascript">

    // ide írjuk a programot (ide kell beszúrni a könyben szereplő példaprogikat!)

  </script>
  </body>
</html>
```

1. példa

JavaScript programunkat (innentől **JS**-nek nevezem) a mintának megfelelően mindig a **<script></script>** rész közé kell írnunk. Ez a további program-mintákban már nem szerepel. A mintapéldák „*kopi-pészt*” beilleszthetőek és kipróbálhatóak. Fontos, hogy a **JS** az utasításaiban különbséget tesz kis és nagybetű között. Figyeljünk erre nagyon!

A képernyőre (a böngésző vásznára) közvetlenül a **document.write** utasítással tudunk kiírni. Minden utasítást pontosvesszővel kell lezárni. A *dupla perjel* csak megjegyzés a programban, a fordító figyelmen kívül hagyja.

```
document.write('Hello World!');
// A megjeleníteni kívánt szöveget aposztrófok közé kell tenni.
```

2. példa

Természetesen a szövegben alkalmazhatunk html formázást is. *A 3. példa* több adat egyidejű megjelenítését is mutatja:

```
document.write('<b>Hello</b> World!<br>');
document.write('<i>Helló világ ez itt a második sor! </i><br>');
document.write('harmadik sor', ' ez is a harmadik sor<br>');
/* A megjeleníteni kívánt szöveget aposztrófok közé kell tenni,
  így több soros megjegyzéseket (kommenteket) is fűzhetünk a progihoz,
  csak ne feledjük el lezárni!
*/
```

3. példa

Az aposztrófok közé írt szöveget a program megjeleníti. Ha elhagyjuk az aposztrófot, akkor a program értelmezni próbálja a beírtakat, és a kifejezés eredményét jeleníti meg:

```
document.write('<b>1+1=</b>',1+1,'<br>');
/* persze ne keressünk túl mély értelmet a progiban, hiszen a következő sor
is ugyanazt az értéket számolja ki ☺ */
document.write('<b>2+1=</b>',1+1,'<br>');
```

4. példa

Ebből következik, hogy tetszőleges számtani műveletet el tudunk végezni:

```
document.write(1548*23+15/14-12, '<br>');
// persze a JS ismeri a zárójelet, így a köv. sor egészen más eredményt hoz:
document.write(1548*(23+15)/(14-12), '<br>');
/* precedencia szabálynak hívják – randa név, de mit tehetünk …
   az alkalmazott műveleti jeleket pedig aritmetikai operátoroknak nevezzük – ez
   sem  szebb.  Itt van még egy: */
document.write('100 osztva 8cal a maradék=',100%8, '<br>');
/* a százalékjellel végzett művelet neve: Modulo, és az osztás maradékát adja.
   Fura, de gyakran lesz rá szükség. */
```

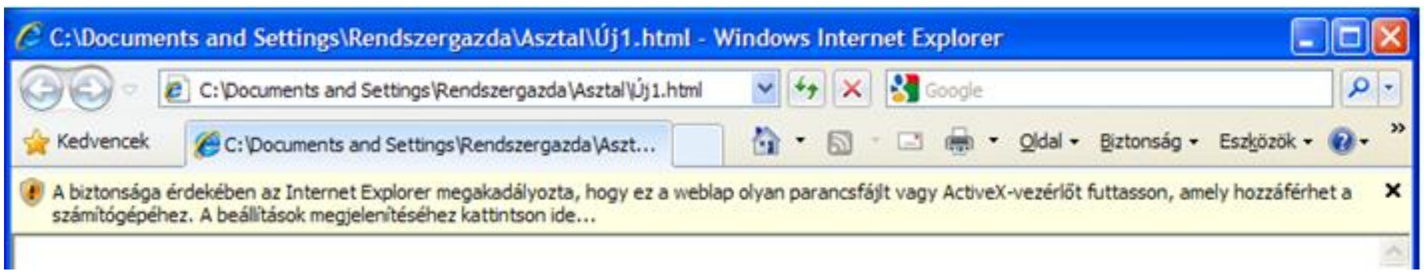
5. példa

2. lecke – „Bogarak” a programban, avagy hibakeresés

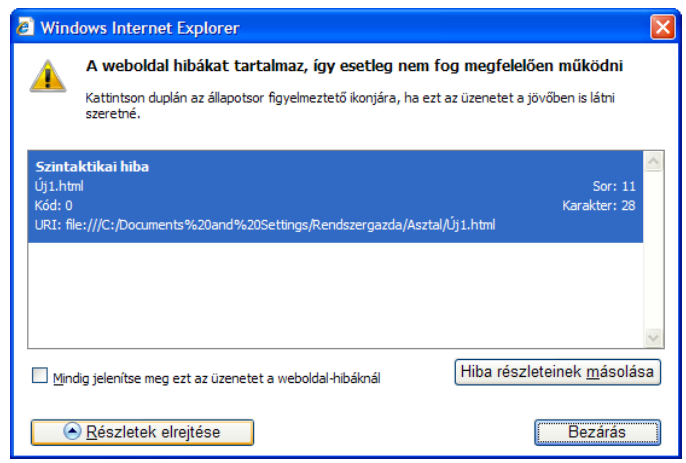
A következő mintapéldában számos programhibát vétettem. Lássuk, milyen segítséget kaphatunk a hibák megtalálásához.

```
document.write(1+1, '<br>);
document.write(1+1, '<br>')
documentum.write(1+1, '<br>');
```

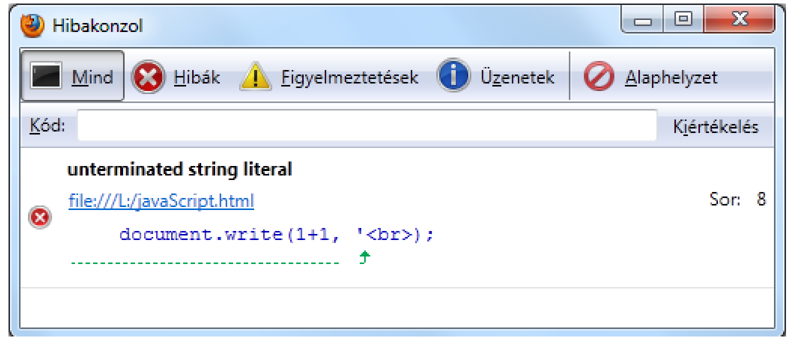
A böngészőben futtatva a programot, fehér vászon fogad minket. Semmi hibaüzenet. Nézzük böngészőnként a hibakeresést. Az **Internet Explorer** szerint veszélyes programozók vagyunk, akiktől jobb óvakodni:



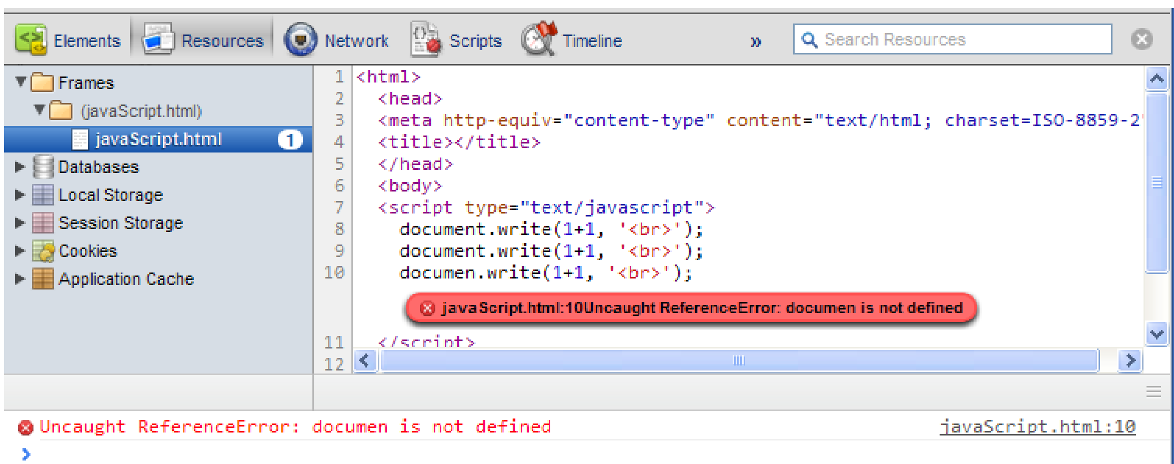
A böngésző alsó sarkában felkiáltójel figyelmeztet a hibára. Rákattintva kiírja, hogy hányadik sorban nem tudja értelmezni az utasítást. Szerintem ne háborgassuk tovább ezt a böngészőt, keressünk egy barátságosabbat (bocs Bill…)



Mozilla Firefox böngészőben kattintsunk az *eszközök /hibakonzol* sorra. A felugró ablak szépen mutatja a forráskódban lévő szintaktikai hibát. A hibaablak sajnos az előző programok hibáit is gyűjti, így lehet, hogy a lista végén találjuk a ránk vonatkozó üzenetet. Egyszerűbb az *„Alaphelyzet”* gomb segítségével kitörölni minden üzenetet, majd frissítés után már csak a ránk vonatkozó sor látszik.



Google Chrome esetén a „menü”-re katt, majd *eszközök/Java Script konzol* (egyszerűbb:CTRL-SHIFT-J) a képen látható ablak alsó részén megjelenik a hibás sor (ill. hibás sorok), a sorra kattintva a böngésző szépen kijelöli a hibás részt, aláírja, hogy mi a panasza. Itt például a 10. sorban nem érti a *„documen”* kifejezést. A felette lévő két sor már javított. Hasonlítsuk össze a hibásat a már javítottal.



Az itt elkövetett hibákat **szintaktikai hibának** nevezzük. Ez azt jelenti, hogy nem a gondolkodásunkban volt a hiba (az a *szemantikai hiba* lesz), hanem a nyelvet alkalmaztuk helytelenül. A programozók munkájuk során jelentős időt töltenek el programhibák keresésével, amit angolul **debugging**-nak (bogarászásnak) neveznek.

3. lecke – „Phanta rei”, avagy változók a programban

```
document.write(kutya,'<br>');
```

6. példa

Ez a példa hibaüzenetet eredményez (*kutya is not defined*), vagyis a fordító próbálja értelmezni a *kutya* kifejezést (mint az *1+1*-et), de nem sikerül neki. Akkor magyarázzuk meg a programnak, hogy mit értünk ez alatt:

```
var kutya= 101;                          // ez egy fiók, amibe egy szám került
document.write(kutya        ,'<br>');   // most a 101 szám jelenik meg
document.write(kutya+1      ,'<br>');   // most a 102 szám jelenik meg
document.write(kutya*2      ,'<br>');   // most a 202 szám jelenik meg
document.write(kutya*kutya, '<br>');    // most a 101 szám négyzete jelenik meg
```

7. példa

A **var** kutya= '101'; utasítással (*var: variant = változó*) létrehoztunk egy fiókot, amibe egyből értéket is tettünk. Ez a fiók addig őrzi meg a tartalmát, amíg felül nem írjuk egy újabb értékkel, pl. így: *kutya=42;* Az értékadással azt is megmondtuk, hogy milyen típusú adatot kívánunk tárolni. A **JS** három elemi adattípust ismer: szám, karakteres (*string*-nek nevezzük) és logikai.

```
var a = 101;                            // ez egy szám (number) típusú változó
var f = 3.1415926;                      // ez is szám (tizedesPONT!)
var n = 1e4;                            // ez egy szám normálalakja (10000)
var b = 'kutya';                        // ez egy karakteres változó – aposztróf!
var c = 'füle';                         // ez is egy karakteres változó
var d = false;                          // ez egy logikai változó
document.write('K= ',2*a*f,'<br>');     // művelet két számtípussal
document.write(b     ,'<br>');          // string kiírása
document.write(b+c   ,'<br>');          // két string összefűzése
document.write(d, ' ',!d,'<br>');       // logikai változó és ellentettje -!negálás
```

8. példa

Egy változóval bármilyen műveletet végezhetünk, az egészen addig megőrzi értékét, amíg egy értékadó operátorral meg nem változtatjuk annak tartalmát (***lásd: „értékadó operátorok" a könyv végén***). A **JS** több értékadó operátort is ismer. A leggyakoribb persze az egyenlőségjel. Nézzük a lehetőségeket:

```
var a = 42;                              // ez egy szám (number) típusú változó
var b =  5;                              // ez is szám
var c;                                   // ez egy nem definiált típusú változó
document.write('c=', c, '<br>');         // naugye, hogy nem definiált
document.write(a*2, ' ', a, '<br>');     // a értéke továbbra is 42
a = a + 1 ;                              // a értékét növeld+1, így "a" értéke=43
document.write(a, ' ', a*2, '<br>');     // a értéke továbbra is 43
++a;                                     // ez is értékadó op. Jelentése: a=a+1
document.write(a, ' ', a*2, '<br>');     // a értéke 44
document.write(++a, '<br>');             // a értéke 45! Először növeli, majd kiírja
document.write(a++, '<br>');
/* a értéke 46, de 45-öt ír ki!
   Ugyanis először kiírja majd utána növeli a változó értékét */
document.write(a, '<br>');
--b ;                                     // ez is értékadó operátor jelentése: b=b-1
document.write(b, '<br>');               // b értéke 4
document.write(b--, '<br>');
/* b=3, de 4et ír ki! Először kiírja, majd utána növeli a változó értékét */
document.write(--a * ++b, '<br>');       // Ez mennyi lesz ???
a=a+b;                                   // a változó új értéket kap
a += b;                                   // ez ugyanaz az utasítás: a= a+b
c=2;
c-=a;                                    // c = c-a
document.write(c, '<br>');               // ki tudod fejben számolni?
```

9. példa

Tehát az **a=a+1**, valamint a **++a** utasítás ugyanaz, csupán kényelmesebb, gyorsabb leírni az utóbbit. Gyakran van szükség egy változó értékének növelésére, csökkentésére, ezért kapott külön operátort ez a kifejezés.

4. lecke – „Jaj! Valami ördög… vagy ha nem, hát… kisnyúl..”, avagy feltételek a programban

Egy derék holland matematikus (kimondhatatlan a neve:-) szerint a strukturált programok három vezérlőszerkezetből épülnek fel: **szekvencia** (utasítások egymás utáni végrehajtása), **szelekció** (feltételes elágazás) és **iteráció** (feltételes utasítás-ismétlés). Eddigi programunkban csak szekvenciákat használtunk, most a szelekció következik.

Döntsük el egy számról, hogy páros, vagy páratlan! A kiértékeléstől függően írjuk ki a „páros”, vagy „páratlan” szöveget. (E*gy szám páros, ha kettővel osztva a maradék = 0*)

```
var a = 42;                                  // ez egy szám (number) típusú változó
if (a % 2 == 0) {                            // a zárójelen belül egy logikai állítás
   document.write(a, ' szám páros<br>');     // akkor fut le, ha az állítás IGAZ
}                                            // itt van vége a feltételnek
```

10. példa

Az **if** utasítás után zárójelben egy olyan állítást kell megadnunk, amelynek logikai eredménye lesz (igaz/hamis). Az **a % 2** művelet után álló „==” jelentése: *egyenlő-e*? Ne keverjük össze az „=” jellel, amely értékadást takar. Ez egy kérdés, amely igaz, vagy hamis (***lásd: „relációs operátorok” függelék***). A feltétel után írt **„{ }"** jelek közé írt utasítások akkor futnak le, ha a feltétel igaz, ellenkező esetben a fordító átugorja a blokkot (a kapcsos zárójel közé tett részt). A változó értékét páratlanra módosítva a mintaprogram nem ír ki semmit. Oldjuk meg a páratlan érték felismerését:

```
var a = 42;                                     // szám típusú változó
if (a % 2 == 0) {                               // logikai állítás
   document.write(a, ' szám páros<br>');        // akkor fut le, ha az állítás IGAZ
} else {
   document.write(a, ' szám páratlan<br>');     // akkor fut le, ha az állítás HAMIS
}                                               // itt van vége a feltételnek
```

11. példa

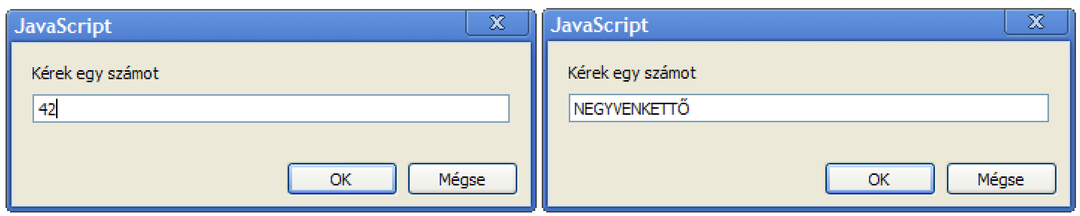
Az **else** után írt blokkban lévő utasítások akkor futnak le, ha az **if** után írt fetétel hamisnak bizonyult.

A **prompt** utasítás segítségével futási időben kérhetünk be a felhasználótól adatot.

```
var a = prompt('Kérek egy számot','0');     // változó bekérése futási időben
if (a % 2 == 0) {                           // logikai állítás
   document.write(a, ' szám páros<br>');    // akkor fut le, ha az állítás IGAZ
} else {
   document.write(a, ' szám páratlan<br>'); // akkor fut le, ha az állítás HAMIS
}                                           // itt van vége a feltételnek
```

12. példa

A programozók munkájának jelentős részét teszi ki a felhasználótól kapott adatok helyességének ellenőrzése. Magyarul olyan program írása, amelyet nem lehet hibásan megadott adatokkal tönkretenni. Ha a beviteli mezőbe szám helyett stringet írunk be, akkor a program szerint ez páratlan. A helyes megoldás az lenne, ha először ellenőriznénk, hogy számot írt-e be a kedves felhasználó!




```
var a = prompt('Kérek egy számot','0');        // változó bekérése futási időben
if (isFinite(a)) {                              // ha "a" szám típusú, akkor…
   if (a % 2 == 0) {                           // ha "a" osztató kettővel, akkor…
      document.write(a, ' szám páros<br>');    // akkor fut le, ha az állítás IGAZ
   } else {
      document.write(a, ' szám páratlan<br>');// ha az állítás HAMIS
   }                                           // vége a páros/páratlan feltételnek
} else {                                       // ha nem számot írt be
   document.write(a, ' NEM SZÁM!<br>');
}                                              // vége a feltételnek
```

13. példa

Az **isFinite()** függvény (***lásd: függelék „globális függvények”***) logikai **igaz** értéket ad vissza, ha a paramétere szám típus. Korlátozzuk a bekért számot egész számra (nem egész számok esetén nem sok értelme van a páros/páratlan fogalomnak).

```
var a = prompt('Kérek egy egész számot','0'); // változó bekérése futási időben
if (isFinite(a)) {                            // ha "a" szám típusú, akkor…
   if (Math.floor(a)==a) {                    // ha egész szám, akkor…
      if (a % 2 == 0) {                       // ha "a" osztható kettővel
         document.write(a, ' páros<br>');     // akkor fut le, ha az állítás IGAZ
      } else {
        document.write(a, ' páratlan<br>');   // ha az állítás HAMIS
      }                                       // vége a páros/páratlan feltételnek
   } else {                                   // ha nem számot írt be
     document.write(a, ' NEM EGÉSZ!<br>'); }  // ha nem egész számot írt be
 } else {
   document.write(a, ' NEM SZÁM!<br>');
}                                             // vége a feltételnek
```

14. példa

A **Math.floor()** függvény (***lásd: függelék „Math objektum”***) a paraméterként kapott szám egészrészét adja vissza. Egy szám pedig nem egyenlő az egészrészével, ha a szám nem egész (48.8 != 48).

5. lecke – Összetett feltételek és logikai állítások

Kérjünk be a felhasználótól egy 100-nál nagyobb páros számot:

```
var a = prompt('Kérek egy 100-nál nagyobb páros számot','0');// változó bekérése
if (a % 2 == 0) {
   if (a > 100) {
      document.write(a, ' OK<br>');
   }
}
```

15. példa

Az eddig tanult beágyazott **if** segítségével a feladat könnyedén megoldható. Írjuk le egyetlen **if**-fel a feltételt:

```
var a = prompt('Kérek egy 100-nál nagyobb páros számot','0');// változó bekérése
if (a % 2 == 0 && a > 100)  {         // ha "a" páros ÉS nagyobb mint 100, akkor…
   document.write(a, ' OK<br>');
} else {
   document.write(a, ' NEM OK<br>');
}
```

16. példa

Az **&&** jel az **ÉS** logikai műveletet jelenti (***lásd: függelék „logikai operátorok”***), amely akkor igaz, ha mindkét feltétel igaz. Tehát hiába lesz *igaz* az egyik logikai állítás, ha a másik *hamis,* akkor az egész *állítás* hamis lesz.

A	B	A && B
H	H	**H**
H	I	**H**
I	H	**H**
I	I	**I**

Az **||** jel a **VAGY** logikai műveletet jelenti, amely akkor igaz, ha már az egyik állítás igaz. Próbáljuk ki a programot VAGY operátorral is. Elfogadja a 6-os, a 101-es, a 120-as számot, de a 7-es már nem jó. Miért?

A	B	A || B
H	H	**H**
H	I	**I**
I	H	**I**
I	I	**I**

Kérjünk be kettővel, **vagy** héttel osztható 100-nál nagyobb számot.

```
var a = prompt('Kérek 2-vel,vagy 7-el osztható 100-nál nagyobb számot','0');
if (a % 2 == 0 || a%2 == 7 && a > 100) {
   document.write(a, ' OK<br>');
} else {
   document.write(a, ' NEM OK<br>');
}
```

17. példa

Úgy működik, ahogy szerettük volna? *Hoppá, elfogadja a kettőt, holott nem nagyobb száznál! Hogyan lehetséges?* A válasz az, hogy a logikai műveletek ugyanúgy nem a beírt sorrendben hajtódnak végre, mint a hagyományos algebrai műveletek: **15+2*42**. Először a 2*42 műveletet kell elvégezni.

Tehát az **a%2 ==0 VAGY a %7==0 ÉS a > 100** esetén először az **a %7==0 ÉS a > 100** művelet értékelődik ki. Programunk tehát 7-tel osztható 100-nál nagyobb számokat VAGY bármely kettővel oszthatót elfogad.

Mi a megoldás? : **(15+2)*42**. Bizony, a zárójel segítségével már mi szabályozhatjuk a műveletvégzés sorrendjét.

```
var a = prompt('Kérek 2-vel,vagy 7-el osztható 100-nál nagyobb számot','0');
if ( (a % 2 == 0 || a%2 == 7) && a > 100) {
   document.write(a, ' OK<br>');
} else {
   document.write(a, ' NEM OK<br>');
}
```

18. példa

Most fordítsuk meg az eredeti logikai feltételt: *Olyan kettővel és héttel sem osztható számot fogadjunk el, amely 100-nál kisebb.* Bonyolultnak tűnik? Pedig a megoldás nem az:

```
var a = prompt('Kérek 2-vel, és 7-el sem osztható 100-nál kisebb számot','0');
if (!((a % 2 == 0 || a%2 == 7) && a > 100)) {
   document.write(a, ' OK<br>');
} else {
   document.write(a, ' NEM OK<br>');
}
```

19. példa

A felkiáltójel a logikai tagadás jele. Az előző feladat logikai állításának értékét fordítottuk meg. A tagadást a programozók negálásnak hívják (a nem programozók nem hívják negálásnak). A negálás művelet igazságtáblája a következő:

A	!A
H	**I**
I	**H**

(Persze van más megoldás is összetett logikai műveletek negálására, amelyhez a [de Morgan] azonosságokra van szükségünk – de erről majd később….)

6. lecke –"Egy, megérett a meggy, kettő, csipkebokor vessző", avagy a számláló ciklus

Számoljunk el 0-tól 20-ig, és az értékeket jelenítsük meg:

```
for (i=0; i <=20; ++i) {
   document.write(i, '<br>');
}
```

20. példa

A **for** egy számláló ciklust megvalósító utasítás (iteráció). A blokkban (kapcsos zárójelek) elhelyezett utasítások ebben a példában 21x fognak ismétlődni. Az "i" változó reprezentálja a tényleges számlálást.

- Első futáskor az "i" változó 0 (**i=0;** első paraméter)
- Minden egyes ciklus ismétléskor "i" változó értéke egyel növekszik (**++i;** harmadik paraméter)
- A ciklus addig fut, amíg i kisebb egyenlő 20 (**i<=20;** második paraméter)
- A ciklusmagban a ciklusváltozót (*példánkban "i"-t*) **TILOS** megváltoztatni! (*ha nem hiszed, próbáld ki...!*)

Számoljunk el 0-tól 20-ig kettesével, valamint írjuk ki a ciklusváltozó "i" négyzetgyökét:

```
for (i=0; i <=20; i+=2) {                    // i+=2  -- > i=i+2
   document.write(i, ' ', Math.sqrt(i), '<br>');
}
```

21. példa

Számoljunk el 20-tól 0-ig visszafelé:

```
for (i=20; i >=0; --i) {
   document.write(i, '<br>');
}
```

22. példa

Keressük meg 1 és 100 között az összes héttel osztható számot

```
for (i=1; i <= 100; ++i) {
   if (i%7==0){document.write(i, '<br>');}   // csak akkor írja ki, ha osztható
}
```

23. példa

Az előző feladatnak van egy másik megoldása is. Használjuk ki a *21. példában* tanultakat:

```
for (i=1; i <= 100; i+=7) {       // Most hetesével számolunk…
   document.write(i, '<br>');
}
```

24. példa

Itt nem kell feltétel, hiszen az "i" változóhoz mindig hetet adva az osztható lesz héttel. Ez a két példa szépen szemlélteti, hogy egy feladat többféleképpen is megfogalmazható. A programozó feladata a leghatékonyabb (legszebb, leggyorsabb) algoritmus megtalálása.

Számoljuk meg, hogy 1 és 10000 között hány héttel osztható szám van. Tehát most nem az értékekre vagyunk kíváncsiak, hanem a mennyiségükre. A programozók ezt az algoritmust a "*megszámlálás tétele*" néven emlegetik.

```
var db = 0;                     // Ebben a változóban számoljuk meg a mennyiséget
for (i=1; i <= 10000; ++i) {
   if (i%7==0) { ++db; }        // Ha a feltétel igaz, akkor növeli a változót
}
document.write('1 és 10000 között ',db,' darab héttel osztható szám van.');
```

25. példa

Adjuk össze 1 és 100 között a számokat. Mennyi az eredmény? Az algoritmus neve: az "*összegzés tétele*".

```
var szum = 0;                       // Ebben a változóban számoljuk meg a mennyiséget
for (i=1; i <= 100; ++i) {
    szum += i;                     // szum = szum + i.
}
document.write('1 és 100 között a számok összege=', szum, '<br>');
```

26. példa

Itt a megoldás, hogy minden ciklus ismétlésnél a "*szum*" változó értékéhez hozzáadjuk az aktuális "*i*" változó értékét. A "+=" jel csak egy egyszerűsített operátor. Egyenértékű a "*szum = szum + i"* kifejezéssel (*9. példa*).

Egy [derék német matematikus](#) kisdiák korában elegánsabb megoldást talált adott számintervallum (számtani sorozat) összegének meghatározására.

Írjuk ki 1..100-ig az összes egész számot, négyzetét, négyzetgyökét táblázatos (!) formában.

```
var s = '';                          // üres, karakter típusú változó
for (i=1; i <= 100; ++i) {
   s += '<tr><td>'+i+'</td><td>'+i*i+'</td><td>'+Math.sqrt(i)+ '</td></tr>';
}
document.write('<table border="1px">',s,'<table>');
```

27. példa

Sok újdonság van a programban. Először is a "**+**" operátorok itt nem összeadás műveletet, hanem hozzáfűzést jelentenek (*8. példa!*) Két karakteres típusú, vagy karakter és szám típus egymás mellé illesztését jelenti. Az "**s+=**" kifejezés jelentése: *s=s+'új adatok'*, tehát gyűjtés. A html tag-ek a **JS** számára közönséges szövegek, majd a *html fordító* fogja kódként értelmezni. A teljes táblázat szintaktika a "*document.write*" sorban áll össze.

7. lecke – "Előbb lövünk, aztán kérdezünk", avagy az elől- és hátul tesztelő ciklusok

Kérjünk be a felhasználótól egy számot. Sikertelen adatbevitel esetén (nem számot adott meg) ismételjük meg újra és újra az adatbekérést.

Az látszik, hogy a feladat ciklussal oldható meg. Azonban a számláló ciklus erre alkalmatlan, mivel nem egyértelmű, hogy hányszor fogja elrontani a felhasználó az adatbevitelt (valószínű, hogy elsőre sikerül neki…). Új ciklus-szervező utasítással kell megismerkednünk: A **while** ciklussal.

```
var n;                          // a változónak még nincs értéke (így típusa sincs)
while (!isFinite(n)) {          // addig ismétlődjön a ciklus, amíg "n" nem szám típusú
  n = prompt('Kérek egy számot','0');
}
document.write('A megadott szám=', n);
```

28. példa

A **while** ugyanúgy működik, mint az **if** utasítás, azzal a különbséggel, hogy a *ciklusmagban* (a kapcsos zárójelek közti részben) írt utasításokat újra és újra ismétli, addig, amíg a logikai állítás *igaz*. Mi történik akkor, ha az állítás mindig igaz marad? Akkor írtunk egy végtelen ciklust. Ebből logikusan következik az, hogy a ciklusmagban kell lennie legalább egy utasításnak, amely ezt a feltételt megváltoztatja. (Amikor egy számítógépről megállapítjuk, hogy az lefagyott, akkor valójában a gépen futó programok közül valamelyik végtelen ciklusba került, amiből többé nem tud kikeveredni.)

Az előző programon hajtsunk végre egy pici módosítást:

```
var n=42;                               // a változónak VAN értéke és az szám típusú
while (!isFinite(n)) {
  n = prompt('Kérek egy számot','0');
}
document.write('A megadott szám=', n);
```

29. példa

Nocsak, elromlott a program! Miért nem kér be adatot? Miért írja ki kapásból, hogy *a megadott szám=42*? Nem romlott el semmi. Olvassuk csak ki a megadott feltételt: "*Addig ismétlődjön a ciklus, amíg n nem szám típusú*". Ez történt. Már az első futás előtt szám típusú volt, így a ciklus egyszer sem futott le (mondtam, hogy olyan mint az **if** utasítás) Ezt a ciklus szerkezetet **elöl tesztelő** ciklusnak hívjuk.

Akkor biztos van hátul tesztelő is. Van. Íme:

```
var n=42;                                      // a változónak VAN értéke és az szám típusú
do {                                           // ciklus kezdete. Nincs feltétel.
  n = prompt('Kérek egy számot','0');
} while (!isFinite(n));                        // a végén értékel, hogy kell-e ismételnie
document.write('A megadott szám=', n);
```

30. példa

Így már bekéri a számot a program, még akkor is, ha adtunk meg kezdőértéket. A ciklus a végén tesztel, tehát ezzel a megoldással a ciklus tartalma egyszer mindenképpen lefut.

Összefoglalva, az elől- és hátul tesztelő ciklus közt a lényegi különbség, hogy az előbbi "*0 és végtelen*", az utóbbi "*1 és végtelen*" között futtatja a ciklusmagot.

Készítsük el újra a **14. példa** feladatát. A cél egy pozitív egész szám bekérése a felhasználótól. Most viszont ciklus segítségével írjuk meg. Tehát újra és újra megjelenik a párbeszéd ablak, addig, amíg a felhasználó minden feltételnek megfelelő számot nem ad meg (programozási példákban gyakori feladat a felhasználó által megadott adatok ellenőrzése). A feladatot hátul tesztelő ciklussal oldjuk meg (természetesen elöl tesztelőssel is megoldható: *lásd: 28. példa*).

```
var n=0;
var ok = false;                           // logikai változó
do {
  n = prompt('Kérek egy pozitív egész számot','0');
  if (isFinite(n)) {
           if (Math.floor(n) == n){
              if (n > 0) {
                 ok = true;}}}
} while (!ok);                           // addig ismétel, amíg nem ok!
document.write('A megadott szám=', n);
```

31. példa

Most oldjuk meg egyetlen "**if**"-el. Tudjuk, hogy az egymásba ágyazott feltételek logikai ÉS művelettel is megoldhatóak. Tehát…

```
var n=0;
var ok = false;                           // logikai változó
do {
  n = prompt('Kérek egy pozitív egész számot','0');
  if (isFinite(n) && Math.floor(n)==n && n > 0) {ok = true;}
} while (!ok);                           // addig ismétel, amíg nem ok!
document.write('A megadott szám=', n);
```

32. példa

Oldjuk meg "**if**" nélkül.

```
var n=0;
var ok = false;                                        // logikai változó
do {
  n = prompt('Kérek egy pozitív egész számot','0');
  ok = (isFinite(n) && Math.floor(n)==n && n > 0); // logikai kifejezés
} while (!ok);                                        // addig ismétel, amíg nem ok!
document.write('A megadott szám=', n);
```

33. példa

Következő programunk egy számra fog "gondolni". A mi feladatunk az, hogy a gép által gondolt számot kitaláljuk. Minden programozási nyelvben megtalálható a "véletlen szám" előállító függvény, amely 0..1 intervallumban választ egy számot. Ezt az értéket egy kis számtani bűvészkedés segítségével tetszőleges intervallumra kiterjeszthetjük.

```
var n = Math.floor(Math.random()*100)+1;
var tipp = 0;
var szoveg = 'Gondoltam egy számot. Tipp?';
do {
     tipp = prompt(szoveg, tipp);
     if (tipp > n) { szoveg = 'Kisebb!';}
     if (tipp < n) { szoveg = 'Nagyobb';}
} while (n != tipp);
document.write('Gratula!....');
```

34. példa

Hogyan lehetne ezt a programot „megfordítani”? Azaz mi gondolunk egy számra és gép próbálja kitalálni az általunk gondolt számot.

8. lecke – Matematikai algoritmusok

Prímszám kereső algoritmus

Döntsük el egy pozitív egész számról, hogy prímszám e! Bizony nem is olyan egyszerű erre a feladatra megfelelően hatékony algoritmust írni. Első próbálkozásunk, hogy elosszuk a vizsgált számot *2 és a szám fele* közti összes számmal. Ha talált osztót (n%i==0), akkor a szám nem prím. (Most számtípus ellenőrzést nem tartalmaz a kód)

```
var n = prompt('Kérek egy számot!',0);
var veg = Math.floor(n/2);                    // szám felének egészrésze
var prim_e = true;
var i = 2;
while (i <= veg && prim_e ) {                 // futás "vég"-ig, vagy amíg nincs osztó
   if (n % i== 0) { prim_e = false;}          // talált osztót…
   ++i;                                       // következő osztó
}
if (prim_e) { document.write(n, ' prímszám ', i, ' osztás');}
else        { document.write(n, ' NEM prímszám ', i, ' osztás');}
```

35. példa

Hatékony az algoritmus? Próbáljuk ki ezt a két prímet: **479001599**; **87178291199**; Nos? Az elsőre 239500800 osztás után (kb. 1 perc) közli, hogy prím. És a második? Bocs, hogy lefagyasztottam a géped! (Nem volt türelmem kivárni, így nem tudom ideírni, mennyi ideig tartana). Egy trükkel tudjuk csökkenteni az osztások számát. Vizsgáljuk meg például a 36 összes osztóját (2*18; 3*12; 4*9; 6*6). Látható, hogy egy számnak osztópárjai vannak. Vannak alsó osztók (pl.:2) és felső osztók (pl.: 18). Az alsó és felső osztók határa a szám gyöke! Elég egy szám alsó osztóit megvizsgálni, ha ott nincs osztó, akkor már nem is lesz. Így elég "**veg"** változó értékét lecserélni, és már kész is. Próbáljuk ki újra, a fentebb említett két prímszámot. Ugye gyorsabb lett. *Hány osztás volt?*

```
var veg = Math.floor(Math.sqrt(n));           // szám gyökének egészrésze
```

Listázzuk ki 2…1000 intervallumban az összes prímszámot.

```
for (n = 2; n <=1000; ++n)    {
  var veg = Math.floor(Math.sqrt(n));    // szám felének egészrésze
  var prim_e = true;
  var i = 2;
  while (i <= veg && prim_e ) {          // futás "vég"-ig, vagy amíg nincs osztó
     if (n % i== 0) { prim_e = false;}  // talált osztót
     ++i;                                // következő osztó
  }
  if (prim_e) { document.write(n, ', ');}
}
```

36. példa

Hogyan tudnánk meghatározni ebben az intervallumban a prímszámok darabszámát, összegét? (Hány prímszám van 2..1000 között? Mennyi az összegük?) A megoldáshoz a 25. és 26. példát kell tanulmányozni.

Számok prímtényezős felbontása

A felhasználótól bekért számot (**n**) bontsuk fel prímtényezőire. A bekért számot osszuk a legkisebb prímmel (**2**), Ha osztható, kiírjuk (**oszto**), majd az új szám az osztás eredménye lesz. Ha nem osztható akkor növeljük az osztót (**+1**). Az eljárást addig ismételgetjük, amíg a szám (**n**) egy lesz.

```
var n = prompt('Kérek egy számot',2);
var oszto = 2;
while (n>1) {                           // ciklus, amíg n nagyobb mint 1
  if (n % oszto == 0) {                 // ha N osztható oszto-val, akkor
      document.write(oszto,'<br>');
      n = n / oszto;                    // n új értéke = n / oszto
} else {
    ++oszto;
}  }
```

37. példa

Legnagyobb közös osztó (LNKO)

Mennyi 2340 és 1092 legnagyobb közös osztója? Több megoldás is létezik a feladat megoldására. Az egyik, hogy bontsuk fel mindkét számot prímtényezőire. Az LNKO pedig a közös prímek szorzata a legnagyobb hatványon.

*Példa: 2340 = 2*2*3*3*5*13; 1092= 2*2*3*7*13; Tehát az LNKO = 2*2*3*13 = 156.*

Ezt a megoldást jelenlegi ismereteinkkel nem tudjuk leprogramozni, hiszen tárolni kell a prímeket (jelenleg nem tároljuk, csak egy változó aktuális állapotát írjuk ki.) Helyette alkalmazzuk az un Euklideszi algoritmust: Osszuk el a nagyobb számot a kisebbel! Az osztás maradékával osszuk el a kisebb számot. A továbbiakban *az osztót osztom a maradékkal* eljárást addig ismétlem amíg 0 maradékot kapok. Az utolsó – 0-tól különböző – maradék a legnagyobb közös osztó.

```
var a = 2340;            // nagyobb szám
var b = 1092;            // kisebb szám
var m = 0 ;
do {                    // hátultesztelő ciklus
   m = a % b;           // m = az osztás maradéka
   a = b;
   b = m;
} while (m != 0) ;      // addig fut, amíg a maradék nem egyenlő nulla
document.write('LNKO=',a);
```

38. példa

Módosuk úgy a programot, hogy "a", és "b" változó értékét a felhasználó adja meg. (Arra is ügyelni kell, hogy "a" változóba a kisebb,"b" változóba a nagyobb érték kerüljön.

Legkisebb közös többszörös (LKKT)

Két szám szorzata egyenlő legnagyobb közös osztójuk, és legkisebb közös többszörösük szorzatával.

LNKO(a,b) * LKKT(a,b) = a*b

Ez hatékony módszert ad a legkisebb közös többszörös meghatározására, mivel elég meghatározni a legnagyobb közös osztót, összeszorozni a két számot, majd a szorzatot elosztani a legnagyobb közös osztóval.

Módosuk az előző programot úgy, hogy kiírja az "LKKT"-t is. Figyelem: "a", és "b" változó értéke módosul a programfutás alatt, így érdemes a program elején a két változót összeszorozni és tárolni.

Háromszög területének számítása három oldalból

```
var a = Number(prompt('-A- oldal',0));    // a Number függvény számmá alakítja
var b = Number(prompt('-B- oldal',0));
var c = Number(prompt('-C- oldal',0));
if (a+b>c && b+c>a && a+c>b  ) { // ha bármely két oldal összege nagyobb
   var k = a+b+c;
   var s = k/2;
   var t = Math.sqrt(s*(s-a)*(s-b)*(s-c));  // by Heron
   document.write('Kerület=',k,' cm<br>');
   document.write('Terület=',t,' cm<sup>2</sup><br>');
   if (a*a+b*b==c*c || a*a+c*c==b*b || b*b+c*c==a*a) {
      document.write('A háromszög derékszögű');
   }
  } else {
     document.write('NEM szerkeszthető háromszög<br>');
}
```

39. példa

Szögfüggvények számítása

Határozzuk meg 0 ..360° között az összes egész fok sinusát, cosinusát, tangensét, cotangensét.

```
for (i = 0; i<= 360; ++i)   {
       var radian = i * Math.PI / 180;   // az értéket radiánban kell megadni
       document.write(i, '° =  radián: ', radian,
                    ' sin= ',  Math.sin(radian),
                    ' cos= ',  Math.cos(radian),
                    ' tan= ',  Math.tan(radian),
                    ' ctg= ',1/Math.tan(radian),'<br>'); // nincs ctg függvény
 }
```

40. példa

Készítsünk a listának szép táblázatos megjelenítést.

9. lecke – Karakterlánc típusok

A karakterlánc típus ASCII karakterek sorozata. A változó értékét aposztófok közé zárva kell megadni. pl így:

```
var s1 = 'Almáspite';                // így egyszerűbb
var s2 = new String('Almáspite');    // … viszont így javasolt megadni
var s3 = new String('');             // Ez egy üres string
var s4 = new String;                 // ….ez is
```

41. példa

Írjuk ki a felhasználótól bekért stringet karakterenként, egymás alá.

```
var s = new String;
s = prompt('Írd be a neved');
document.write('Szia ',s,'<br>');                    // ….udvariasan köszönünk
document.write('A neved ',s.length,' karakterből áll.<br>'); // string hossza

for (i=0; i < s.length; ++i) { // string első karaktere a nulladik elem!
    document.write(s.charAt(i),'<br>');
}
```

42. példa

Nézzük meg alaposan a **for** ciklus paramétereit. A stringet nullától kezdjük el olvasni. Jegyezzük meg a string első karaktere a nulladik. A karakterlánc olvasását pedig a hossz előtt egyel (*i < s.length*) hagyjuk abba. Egy 5 karakter hosszú string olvasása tehát 0-tól négyig tart. A bekért nevet írassuk ki visszafelé (most egy sorban jelenjen meg, így nem kell sortörés)

```
var s = new String;
s = prompt('Írd be a neved');
for (i =s.length-1; i>=0 ; --i) {      // string első karaktere a nulladik elem!
    document.write(s.charAt(i)); }
```

43. példa

A visszafelé számláló for ciklus a *22. példában* már szerepelt.

Ahogy két numerikus típust is össze tudtunk hasonlítani, úgy karakteres típusok is összehasonlíthatóak. Azonban ügyelnünk kell arra, hogy ha a karaktert *s=new String* formában definiáltuk, akkor az összehasonlításnál az *s.valueOf()* alapján kell hasonlítanunk.

```
var s1 = new String('Alma');        // string objektum
var s2 = new String('Körte');
var sA = s1.valueOf();              // String primitív = a string objektum értéke
var sB = s2.valueOf();
if (sA > sB)  { document.write('s1 nagyobb'); }
if (sA < sB)  { document.write('s2 nagyobb'); }
if (sA == sB) { document.write('s1 = s2 ');    }
```

44. példa

Karakterláncok esetén a nagyobb / kisebb fogalom elsőre furcsa lehet, de csupán szoros ABC sorrendről van szó (ASCII kódtábla alapján). Figyelnünk kell arra is hogy kis/nagybetű nagyon nem ugyanaz.

Próbáljunk különböző értékeket adni s1, s2 változóknak, és figyeljük az összehasonlítást!

A következő példaprogram az ASCII kódtábla értékeit jeleníti meg. A 32 előtti értékek nem jelennek meg, mert vezérlő karakterek. (A 32 sem, mert az pedig a szóköz – ami ugyanolyan karakter, mint bármely betű).

```
for (i = 32;  i <= 255; ++ i) {
     document.write(i, '-->',String.fromCharCode(i),'<br>'); }
```

45. példa

A felhasznált "fromCharCode" függvény a paraméterként kapott számhoz (0..255) tartozó ASCII karaktert adja vissza.

A következő példaprogram további string transzformációkat mutat be. Kis- és nagybetűs konvertálást, valamint stringből részlánc kivágását. Fontos megjegyezni, hogy ezek a műveletek a string tartalmát nem változtatják meg. A változó értékének megváltoztatásához értékadó művelet (s=….) szükséges. Az alkalmazott eljárások összefoglalója szintén megtalálható a függelékben.

```
var s = new String('Árvíztűrő tükörfúrógép');

// string kiírása nagybetűs alakban
document.write(s.toUpperCase(),'<br>');

// string kiírása kisbetűs alakban
document.write(s.toLowerCase(),'<br>');

// string részlet kivágása - első paraméter: kezdő pozíció; második: hossz
document.write(s.substr(10,5),'<br>');
document.write(s.substr(15,4),'<br>');

// keresés a stringben (első előfordulás) . ha nincs, akkor -1 -et ad vissza
document.write(s.indexOf('tükör'),'<br>');

// keresés a stringben (utolsó előfordulás) - visszafelé keres
document.write(s.lastIndexOf('r'),'<br>');

var nev = prompt('Kérem a teljes neved!');

// szóköz utáni karakter pozició
var i = nev.lastIndexOf(' ')+1;
document.write('Szia ',nev.substr(i),'!')  ;

// írjuk ki a vezetéknevet nagybetűs alakban
document.write(nev.substr(0,i).toUpperCase(),'<br>');
```

46. példa

Titkosító algoritmusok

Feladatunk egy string tartalmának titkosítása. Persze nagyon egyszerű titkosítás lesz. Ennél egyszerűbb titkosítás nem is létezik. A program a string minden egyes karakterének ASCII kódjához hozzáad egy konstans értéket (*kulcs*), majd a kapott számot visszaváltja karakterré. A karaktersort a "*titkos*" nevű változóba gyűjti. Az algoritmust Caesar kódnak nevezik.

```
var nyilt  = new String('Almáspite');
var titkos = new String('');
var kulcs  = 1;                          // jelszó ☺

for (i=0; i < nyilt.length; ++i) {
    var a = nyilt.charCodeAt(i)+kulcs;   // i. pozició ASCII kódja + kulcs
    var k = String.fromCharCode(a);      // ASCII kód visszaalakítása karakterre
    titkos += k;                         // titkos = titkos + k
}
document.write('nyílt szöveg: ',nyilt,'<br>titkos szöveg: ',titkos);
```

47. példa

Milyen lehetőségeink vannak az így rejtjelezett szöveg visszafejtésére?

A Vigenere kód

Majd jövőre… ☺

10. lecke – Függvények, eljárások

A címben szereplő kifejezés most nem egy algoritmust takar, hanem egy programozási struktúrát, amely nélkül lehetetlen nagyobb és összetettebb programokat írni. Ezt a szerkezetet olyan programrészletekhez tudjuk felhasználni, amelyek gyakran ismétlődnek. Így nevük segítségével bármikor meg tudjuk hívni.

```
function eljaras1 (s) {                         // eljárás neve, paraméter
     document.write(s,'<br>');
}
function eljaras2() {
var i;
   for (i=0; i < nyilt.length; ++i) {
       document.write(i,'.sor<br>'); }
}
var i = 1;                                      // itt kezdődik a program
eljaras1('almáspite');                          // eljárás meghívása, paraméter átadással
eljaras1('pecsenyekacsa');
```

48. példa

A példában szereplő ***eljaras1*** eljárást kétszer is meghívjuk, míg az ***eljaras2*** egyszer sem fut le.

```
function ir (s) {                             // eljárás neve, paraméter
     document.write(s,'<br>');
}
function negyzet (a) {                        // a függvény neve, formális paraméter
  var i = a*a;                                // lokális változó
  return i;                                   // függvény visszatérési értéke
}
var i = negyzet(2);                           // függvény hívása, aktuális paraméter
ir(i);
ir(negyzet(negyzet(negyzet(i))));             // Melyik számot fogja kiírni?
```

49. példa

Ebben a példaprogramban rengeteg újdonság van, amit jól meg kell jegyeznünk. A ***negyzet*** eljárás neve: függvény. Van visszatérési értéke, amelyet a ***return*** utasítással adunk meg. A függvényen (eljáráson) belül létrehozott változók kívülről (a főprogramból, más függvényekből) láthatatlanok, ezeket lokális változóknak hívjuk. Élettartamuk csak a függvény futási idejére terjed. Tehát a függvényben és a főprogramban létrehozott "i" változóknak semmi közük egymáshoz!

Írjunk függvényt, amely a megadott paraméterek alapján kiszámítja az elektromos vezető fajlagos ellenállását.

```
function ir (s) {                        // eljárás neve, paraméter
     document.write(s,'<br>');
}
function vez_ellenallas(d, l, ro) {
   var A = d*d*Math.PI/4;              // a vezeték keresztmetszete  mm^2
   var R = ro*l/A;
return R;   }

var d = 0.1;                           // a vezeték átmérője -  mm
var l = 200;                           // a vezeték hossza - m
var ro = 0.0175;                       // a réz fajlagos ellenállása Ohm*mm^2/m
var ohm = vez_ellenallas(d, l, ro);
ir(d+'mm átm., '+l+'m hosszú rézvezeték ellenállása='+ohm+'Ohm.');
```

50. példa

A függvénynek most három paramétert adtunk át.

Most vizsgáljuk meg a paraméterként kapott évet, hogy szökőév –e?

```
function szokoev(a) {
 return ((a % 4 == 0 && a %100!=0)  || a % 400 == 0);}
var ev = 2001;
var s = ' NEM ';
if (szokoev(ev)) { s = ''}
ir(ev+' év'+s+'szökőév.' );
```

51. példa

11. lecke – Bitműveletek

Bitműveleteket csak fixpontos számábrázolású számokon szabad végezni. (Természetesen ehhez ismerni kell, a fixpontos, valamint a lebegőpontos számábrázolás fogalmát)

```
function ir (s) {
     document.write(s,'<br>');
}
var j = 12;
ir(j<<1)  ;                              // Melyik számot fogja kiírni?
ir(j>>1)  ;                              // Melyik számot fogja kiírni?
for (i=0; i < 31; ++i) { ir(1<<i); }     // kettő hatványai
ir(~j+1);                                // mínusz 12
```

52. példa

A "**<<**" operátor eltolja a biteket balra az operátor után írt számszor. A belépő számjegyek nullák. Ez a művelet megfelel a kettővel való szorzásnak. A "**>>**" operátor eltolja a biteket balra az operátor után írt számszor. A belépő számjegyek nullák. Ez a művelet megfelel a kettővel való egész osztásnak. Ezt használja ki a kettő hatványait előállító (leghatékonyabb) algoritmus is.

Miért ad negatív számot a 2^31 érték? Ha nem tudod a választ, akkor tényleg olvasd el a fixpontos számábrázolást.

A "-12" előállítása szintén a fixpontos számábrázolás kettes komplementerén alapul. (*írtam, hogy olvasd el, de tényleg….*) A szám előtti "hullám jel" minden egyes bitet az ellentettjére pörget (bitenkénti negálás – egyes komplemens).

A következő feladatban egy előre meghatározott bitet kapcsolunk be, majd ki. A műszaki programozási gyakorlatokban jellemző feladat, amikor egy regiszter egyetlen bitjét kell billegtetni úgy, hogy a többi bit ne változzon.

```
function ir (s) {
     document.write(s,'<br>');
}
var j = 78;                      // 1001110 – a 2^4 bitet fogjuk bekapcsolni
var bitmask = 1<<4               // 0010000
j = j | bitmask;                 // 1011110  - bitenkénti "vagy" a bitmaskkal
ir('bekapcsolva: '+j);
j = j & ~bitmask;                // 1001110  - bitenkénti "és" a bitmaskkal negálttal
ir('kikapcsolva: '+j);
```

53. példa

A feladat pontos értelmezéséhez nézd át az ÉS, VAGY igazságtáblákat (5. lecke), valamint a függelék bitművelet operátorait.

Következő függvényünk decimális számokat vált át bináris alakba:

```
function ir (s) {
     document.write(s,'<br>');
}
function dectobin (szam) {
    var binaris = '';
    var szam = 199;
    while (szam != 0) {
        binaris = (szam % 2)+binaris;
        szam = parseInt(szam / 2);                // egészrész
  }
  return (binaris);
  }

  ir( dectobin(199)) ;
```

54. példa

12. lecke – A tömb adatszerkezet

A tömb (*array*) olyan adatszerkezet, amelyet nevesített elemek csoportja alkot, melyekre sorszámukkal (indexükkel) lehet hivatkozni. Ha a változót egy fiókhoz hasonlítottuk, akkor a tömb egy fiókos szekrény.

Jegyezzük meg: *A tömb első eleme a 0. elem!*
A tömb utolsó eleme a length-1. elem!
Adott tömbelemre a tömbnév után tett szögletes zárójelben hivatkozunk: T[3]

```
var ures_tomb = new Array();
var tomb = new Array('Uborka','Dinnye','Krumpli','Dió','Alma','Banán');
document.write('A tömb első eleme=',tomb[0],'<br>');
document.write('A tömb elemszáma=',tomb.length,'<br>');
document.write('A tömb utolsó eleme=',tomb[tomb.length-1],'<br>');

// ------ A tömb elemeinek kiírása ------------------
for (i =0; i<tomb.length; ++i) {
   document.write(i,'. --->',tomb[i],'<br>');
}
document.write('<br>');

// ------ A tömb elemeinek kiírása visszafelé---------
for (i = tomb.length-1; i>=0; --i) {
    document.write(i,'. --->',tomb[i],'<br>');
}
document.write('<br>');

tomb.push('Ribizli');      // új elem beszúrása a tömb végére  (STACK!)
tomb.unshift('Málna');     // új elem beszúrása a tömb elejére (QUEUE!)
tomb.pop();                // utolsó elem törlése
tomb.shift();              // első elem törlése
```

55. példa

A következő példa egy szám hexadecimális konvertálását mutatja be:

```
function ir (s) {
     document.write(s,'<br>');}

function hexa(a) {
  var eredmeny = '';
  var szamjegyek=new Array('0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F');
  while (a!=0) {
    eredmeny = szamjegyek[a%16] + eredmeny;
    a = parseInt(a/16);   }
return eredmeny; }
ir(hexa(282));
```

56. példa

A tömbkezelés további algoritmusai, mint programozási alaptételek kerülnek ismertetésre. Mindegyik programban egy véletlenszámokkal feltöltött tömbbel fogunk dolgozni. Az itt megadott függvényeket így mindegyik program elejére be kell másolni.

```
function ir (s) {document.write(s,'<br>');}

// "a" tömböt "elemszam" darab véletlen értékkel tölt fel "minvel" – "maxvel" közt
function tombFeltolt(a, elemszam, minvel, maxvel) {
      var intervallum = maxvel-minvel;
      for (i=0; i< elemszam; ++i) {
           a[i]=Math.round(Math.random()*intervallum)+minvel;}
    return a; }
}
// "a" tömböt értékeit és pozíció számát (indexét) írja ki
function tombKiir(a) {
     for (i=0; i< a.length; ++i) {
        document.write('(',i,'.)',a[i],', ');}
}
```

57. példa

Tömb elemeinek legkisebb, legnagyobb eleme (minimum, maximum kiválasztás tétele)

Keressük meg a tömb legnagyobb és legkisebb elemét (*A program elejére az előző példa függvényeit be kell másolni!*)

```
var tomb = new Array();
tomb = tombFeltolt(tomb, 100, 1, 1000); // 100 elemű tömb, 1..1000 közti számokkal
tombKiir(tomb);

// -----maximum elem kiválasztás tétele  -----
var max = tomb[0];     // tételezzük fel, hogy a tömb első eleme a legnagyobb…
var poz = 0;           // ebben a változóban a poziciót tároljuk (indexet)
for (i =0; i<tomb.length; ++i) {
     if (tomb[i]>max) { max = tomb[i];  poz = i;}
}
ir('<br>Legnagyobb elem: '+max+', a '+poz+'. pozición.');


// -----minimum elem kiválasztás tétele  -----
var min = tomb[0];   // tételezzük fel, hogy a tömb első eleme a legkisebb
var poz = 0;
for (i =0; i<tomb.length; ++i) {
     if (tomb[i]<min) {min = tomb[i];  poz = i;}
}
ir('<br>Legkisebb elem: '+min+', a '+poz+'. pozición.');
```

58. példa

Fontos, hogy a minimum, maximum értéket tartalmazó változót ne "nulla" kezdőértékkel inicializáljuk, hiszen egyáltalán nem biztos, hogy ez az érték felülíródik (pl. a tömbben csak negatív számok vannak). A jó megoldás, ha a tömb első elemével inicializáljuk, vagy – maximum kiválasztás esetén az ábrázolható legkisebb, - minimum kiválasztás esetén az ábrázolható legnagyobb számot tesszük bele kezdőértéknek.

Tömb elemeinek összege, átlaga (megszámlálás, összegzés tétele)

Számítsuk ki a tömb elemeinek összegét és számtani átlagát. (*A programok elejére az 57. példa függvényeit be kell másolni!*)

```
var tomb = new Array();
tomb = tombFeltolt(tomb, 100, 1, 1000); // 100 elemű tömb, 1..1000 közti számokkal
tombKiir(tomb);
var szum = 0;
 for (i=0; i< tomb.length; ++i) {
    szum = szum + tomb[i];
 }
ir('<br>A tömb elemeinek összege '+szum) ;
ir('<br>A tömb elemeinek átlaga '+szum/tomb.length);
```

59.példa

Az átlagszámítás algoritmusában osztás van. Osztáskor mindig meg kell vizsgálni, hogy a nevezőben lévő változó nem nulla? Példánkban nulla elemű tömb esetén a program hibaüzenettel leállna.

Szekvenciális keresés a tömbben

Feladatunk egy adott érték keresése a tömbben. Két eset lehetséges: az érték többször, vagy csak egyszer fordul-e elő a tömbben? Ha többször is előfordulhat a keresett érték, akkor egyszerű a feladat: végig kell olvasni a tömböt és a feltételnek megfelelő értékek pozícióját ki kell írni (*A programok elejére az 57. példa függvényeit be kell másolni!*)

```
var tomb = new Array();
tomb = tombFeltolt(tomb, 100, 1, 50); // 100 elemű tömb, 1..50 közti számokkal
tombKiir(tomb);
var keres = 42;                       // ezt az értéket keressük
var talalt = false;                   // találatot jelző logikai változó
for (i=0; i< tomb.length; ++i) {
     if (keres == tomb[i]) {
         ir('<br>Találat a(z) '+i+'. pozición');
         talalt = true; } }
if (!talalt) {ir('<br> Nincs '+keres+' a tömbben.');}
```

60.példa

A feladat összetettebb, ha csak egy találat lehetséges, mivel ekkor le kell állítani a keresést. Tehát a "for" ciklus nem alkalmas a feladatra. Elöl tesztelő ciklust használunk, amely "*addig fut, amíg végig nem olvastuk a tömböt és nincs találat*".

```
var tomb = new Array();
tomb = tombFeltolt(tomb, 100, 1, 50); // 100 elemű tömb, 1..50 közti számokkal
tombKiir(tomb);
var talalt = false;
var keres = 42;
var i = 0;
while (i++ < tomb.length && !talalt) {  // ..amíg nincs vége a tömbnek és nincs találat
    talalt = (keres == tomb[i]);    // a zárójelben lévő művelet eredménye logikai típus
}
if (talalt) { ir('<br>Találat a '+(i-1)+' pozición.');}
else        { ir('<br> Nincs '+keres+ ' a tömbben.'); }
```

61.példa

Miért az (i-1) pozíción van találat és miért nem az "i"- pozíción?

Cseréljük le a ciklus-feltétel sort a következőre:

```
while (!(i++ >= tomb.length || talalt)) {  // .amíg nincs vége a tömbnek és nincs találat
```

Működik? Működik. Miért? A válasz a már említett de Morgan azonosság.

A logikai kifejezések között igaz a következő két azonosság: **!A és !B = !(A vagy B)** *valamint* **!A vagy !B = !(A és B)**

Próbáljuk a két logikai feltételt megfeleltetni a "while" után álló logikai kifejezéseknek!

A szekvenciális (soros) keresésnek átlagosan "*elemszám DIV 2*" elemet kell végignéznie, hogy találat legyen. (Néha szerencsénk van és megtalálja elsőre, néha pech és az utolsó rekord lesz a találat.) Ennél hatékonyabb keresés lesz majd a bináris keresés.

Karakteres típusú adatok keresése a tömbben

Ha pontos egyezésre keresünk, akkor nincs lényeges különbség. A html forrás elejére szerkesszük be a következő sort:

```
  <script type="text/javascript" src="j2.js"></script>
// A több mint 2000 települést tartalmazó fájl megtalálható a doksi mappájában "j2.zip" néven
```

```
var keres = 'Zalaegerszeg';
for (i=0; i < telepulesTb.length; ++i) {
    if (keres == telepulesTb[i]) {
              ir(i+' --> '+telepulesTb[i]);
    }
}
```

62.példa

Szépen működik, de karakteres adatok keresésnél elvárjuk a csonkolt keresést. Tehát a `keres="Zala"` kifejezésre nincs találat. Alakítsuk át úgy a programot, hogy megtalálja a "Zala"-val kezdődő településeket. A megoldás, hogy a vizsgált elemet olyan hosszan hasonlítjuk össze, amilyen hosszú a `keres` stringben lévő kifejezés.

```
    if (keres == telepulesTb[i].substr(0,keres.length)) {
```

Karakteres adatoknál célszerű, hogy nem teszünk különbséget a kisbetűk és nagybetűk között. A megoldás hogy mind a keres változóban lévő értéket, mind a vizsgált elemet nagybetűsre (vagy kisbetűsre, lényeg, hogy egyforma legyen) konvertáljuk és így hasonlítjuk össze.

```
    if (keres toUpperCase() == telepulesTb[i].substr(0,keres.length).toUpperCase()) {
```

Tömb elemeinek rendezése (buborékos rendezés)

```
var tomb = new Array();
tomb = tombFeltolt(tomb, 100, 1, 50); // 100 elemű tömb, 1..50 közti számokkal
tombKiir(tomb);
var csere = 0;
var volt_csere = true;
while (volt_csere) {
      volt_csere = false;
      for (i=0; i< tomb.length-1; ++i) {
          if (tomb[i] > tomb[i+1]) {       // ha nagyobb, akkor csere
              volt_csere= true;
              csere     = tomb[i];
              tomb[i]   = tomb[i+1];
              tomb[i+1] = csere;
          }
      }
}
ir('<br>rendezett lista:');
tombKiir(tomb);
ir('<br>A tömb legkisebb eleme: '+tomb[0]);
ir('<br>A tömb legnagyobb eleme: '+tomb[tomb.length-1]);
```

63.példa

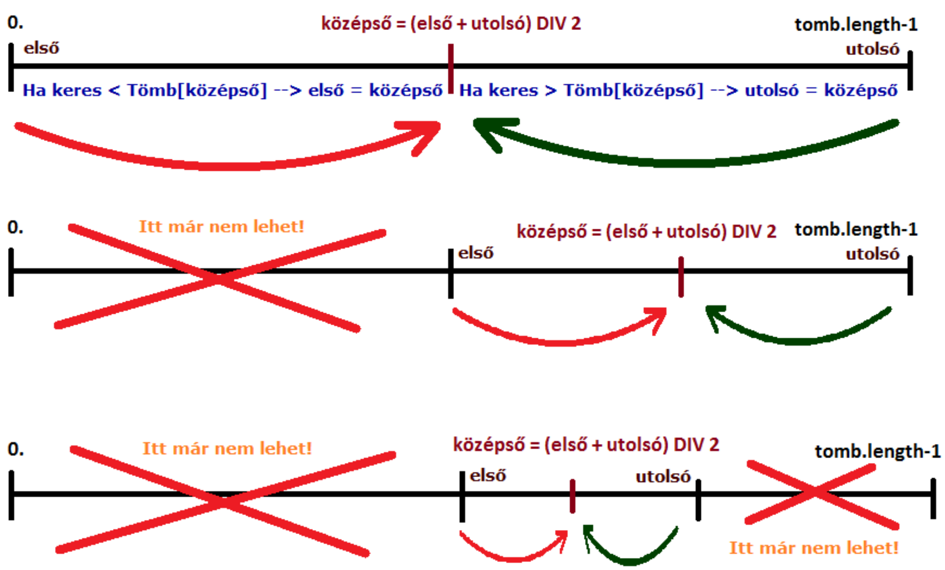
Tömb elemeinek rendezése (minimum, maximum elem kiválasztásos rendezés)

……

Bináris keresés a tömbben (logaritmikus keresés tétele)

Szekvenciális keresésnél átlagosan "*elemszám DIV 2*" értéket kell megvizsgálni, hogy megtaláljuk a kívánt elemet (néha szerencsénk van, elsőre megtalálja, néha pech és az utolsó elem a keresett). A bináris keresés ennél jobb hatásfokkal keres, viszont előfeltétele a rendezett adathalmaz. Az algoritmus menete a következő:

1. tegyük be egy változóba a tömb első elemének indexpozícióját (első = 0)
2. tegyük be egy változóba a tömb utolsó elemének indexpozícióját (utolsó = tomb.length-1)
3. indítsunk egy hátul tesztelős ciklust
    4. középső = első + utolsó DIV 2
    5. Ha a keresett érték kisebb mint a tömb[középső] eleme , akkor első = középső
    6. Ha a keresett érték nagyobb mint a tömb[középső] eleme , akkor utolsó = közép
7. A ciklus addig fut, amíg meg nem találtuk, és nem "ér össze" az első és az utolsó változó értéke



Látható, hogy így minden egyes cikluskör után az előző tömbméret fele marad csak. Így igen nagy sebességgel "elfogy" a tömb.

Mekkora ez a sebesség? Tételezzük fel hogy tömbünk pont 1024 elemű, ekkor a tízedik keresésre "elfogy" a tömb. 2^{10} = 1024, tehát ha az adathalmaz elemszáma **N**, akkor maximum $\mathbf{log_2 N}$ keresésre biztosan véget ér a keresés. Így ez az érték is megadható a ciklus leállási feltételének.

Hány elemet vizsgál meg az eljárás egymillió és egymilliárd rekord esetén?

A html forrás elejére szerkesszük be a következő sort:

```
<script type="text/javascript" src="j2.js"></script>
// A több mint 2000 települést tartalmazó fájl megtalálható a doksi mappájában "j2.zip" néven
```

```
// A bináris kereséshez rendezett tömbre van szükség !
var csere = 0;
var volt_csere = true;
while (volt_csere) {
      volt_csere = false;
      for (i=0; i< telepulesTb.length-1; ++i) {
          if (telepulesTb[i] > telepulesTb[i+1]) {        // ha nagyobb, akkor csere
              volt_csere= true;
              csere             = telepulesTb[i];
              telepulesTb[i]   = telepulesTb[i+1];
              telepulesTb[i+1] = csere;
          }
      }
}
// ---- bináris keresés ----
var keres = 'Pécs';
var elso   = 0;
var utolso = telepulesTb.length-1;
var i = 0;
var leall = parseInt(Math.log(telepulesTb.length)/Math.log(2))+1;
// =log2(rekordszám) ennyi cikluskör lehet maximum
do {
     var kozep = parseInt((elso + utolso) / 2);
     if (keres < telepulesTb[kozep]) { utolso = kozep; }
     if (keres > telepulesTb[kozep]) { elso   = kozep; }
} while ( keres != telepulesTb[kozep]  && i++ < leall );

if (i > leall) { ir('Nincs találat.'); }
else           { ir('Találat a '+kozep+' pozición.');}
```

64.példa

Minden eddig tanult ismereteink összefoglalója következik**: Készítsünk öröknaptárt.** *(A JS-ben ugyan van dátum típusú változó, de mi most matematikai és csillagászati módszerekkel határozzuk meg, hogy a megadott dátum mely napra esik.*) Kiindulási dátumunk 1900.01.01, amely vasárnap volt (tuti). Kiszámítjuk, hány nap telt el azóta, majd osztjuk héttel és a maradékot (0..6) vizsgáljuk. Ha nulla marad, akkor újra vasárnap van, ha egy, akkor hétfő….

```
var hetnapjaTb = new
Array('vasárnap','hétfő','kedd','szerda','csütörtök','péntek','szombat');
var hoNapjaTb  = new Array(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
//---------------------
function szokoev(a) {                // lásd szökőév függvény: 51. példa
//---------------------
  var nap = 0;
  if ((a % 4 == 0 && a % 100 != 0) || a % 400 == 0) { nap = 1; }
return nap;}                         // ha szökőév, akkor 1, ha nem akkor 0

//------------------------------
function naptar(ev, ho, nap) {
//------------------------------
 var napok = 0;
 var i = 0;
 for (i=1900; i < ev; ++i) { napok += 365 + szokoev(i); } // befejezett évek napjai
 for (i=0; i < ho-1; ++i)  { napok += hoNapjaTb[i] ; }    // befejezett hónapok napjai
 napok += nap;                                            // akt. hónapból eltelt napok
 if (ho > 2 ) { napok += szokoev(ev); }           // ha elmúlt február és ez az év szökőév
return hetnapjaTb[ napok % 7  ]; }

//------------------------------
 document.write(naptar(2012,12,21));
```

65.példa

A függvény segítségével készítsük el egy adott év, hónap (pl: 2013, 05) naptárát: Irassuk ki a hónap dátumát, mellé az aktuális napot (egyetlen "for" ciklussal oldjuk meg).

Többdimenziós tömbök

A kétdimenziós tömb tulajdonképpen egy táblázat. A matematikusok "mátrixnak" nevezik. (A nem matematikusok nem nevezik mátrixnak). Ez az adatszerkezet nem más, mint egy olyan tömb amelynek elemei újabb tömbök. Legegyszerűbb értelmezése a sor-oszlop megközelítés. A tömb feldolgozása is így történik, valamint alkalmazása is sor-oszlop értelmezésű feladatokhoz köthető. Klasszikus feladat, egy adott hónap óránkénti hőmérséklet adatainak tárolása (pl 31x24-es tömb). Ebből az adatszerkezetből választ kapunk olyan kérdésekre, hogy:…

- Mennyi volt a hónap átlaghőmérséklete? (teljes tömb összesítés)
- Mennyi volt az egyes napok átlaghőmérséklete? (sorok összesítése)
- Mennyi volt az egyes órák átlaghőmérséklete? (oszlopok összesítése)

```
var tb =                                      /* KÉTDIMENZIÓS TÖMB DEFINIÁLÁSA */
    new Array(
      new Array( 9, 3, 2, 2, 3, 1, 4, 9),
      new Array( 2, 5, 9, 4, 5, 7, 8, 5),
      new Array( 4,10, 4, 4, 6,15, 5, 9),
      new Array( 1,10, 3, 8, 2, 2, 3, 8),
      new Array( 4, 3, 1, 9, 5, 8, 3, 1),
      new Array( 5, 6, 5, 5, 4, 6, 8, 1),
      new Array( 2, 4, 7, 5, 4, 7, 8, 2),
      new Array( 0, 4, 9, 6, 2, 6, 6, 4)
 );
```

66.példa

A kétdimenziós tömb feldolgozásához általában két –egymásba ágyazott- számláló ciklust alkalmazunk. Határozzuk meg a tömb elemeinek összegét és átlagát. Majd ugyanígy határozzuk meg a sorok összegét, átlagát; majd az oszlopok összegét, átlagát:

```
// -- tömb átlaga --
var szum = 0;
for (i=0; i < tb.length; ++i) {
     for (j=0; j<tb[i].length; ++j) { szum=szum+tb[i][j];}
}
document.write('tömb átlag='+(szum/(i*j))+'<br><br>');

// -- tömb kiírása sorok szerint --
for (i=0; i < tb.length; ++i) {
    var szum = 0;
    for (j=0; j<tb[i].length; ++j) {
        document.write(tb[i][j],' ') ;
        szum = szum + tb[i][j];
    }
    document.write(' --> ['+szum+'; '+(szum/j)+']<br>');
}
document.write('<br><br>');

// -- tömb kiírása oszlopok szerint
for (j=0; j<tb[0].length; ++j)  {
    var szum = 0;
    for (i=0; i<tb.length; ++i)  {
        document.write(tb[i][j],' ') ;
         szum = szum + tb[i][j];
    }
    document.write(' --> ['+szum+'; '+(szum/i)+']<br>');
}
```

67.példa

13. lecke – Adatbekérés felhasználótól (HTML)

Eddig a felhasználói adatbekérést a "prompt" eljárással oldottuk meg. A cél, hogy HTML űrlap segítségével kommunikáljunk a programot alkalmazóval. Ehhez több feltételnek is meg kell felelni:

- A HTML űrlap valamely eseményére (pl: kattintás egy nyomógombon) meg kell hívni egy JS eljárást
- A HTML kódban írt értéket át kell adni a JS-nek.

A minta egy HTML űrlapot, valamint az űrlapon lévő nyomógomb kattintás függvényhívási eljárását mutatja be. A mintafeladat érdekessége, hogy a kiírás nem a szokásos „document.write” segítségével történik, hanem egy üres „DIV” –tag-be ír vissza.

```
<script type="text/javascript">
function teszt() {
  var szam1 = parseFloat(document.form1.SZAM1.value); // érték átvétele HTML-ből!
  var szam2 = parseFloat(document.form1.SZAM2.value); // parseFloat: Szám típusra alakít!
  document.getElementById('keret1').innerHTML = 'Összegük='+(szam1+szam2)+
  '<br>Különbségük='+(szam1-szam2)+ '<br>Szorzatuk='+(szam1*szam2)+
  '<br>Hányadosuk='+(szam1/szam2)+'<br>Hatványuk='+Math.pow(szam1,szam2);
return true; }

</script>

<form name="form1">
   1. szám: <input type="number" name="SZAM1" value="0">  <!-- "number" HTML5 alatt jó -->
   2. szám: <input type="number" name="SZAM2" value="0">
   <input type="button" value="OK" onClick="return teszt();">
   <!-- A nyomógombra kattintva lefut a "teszt" JS függvény -->
</form>
<div class="A0" id="keret1"></div>
<!--  ebbe az üres DIV-be írja be a "teszt" függvény az eredményeket (innerHTML)  -->
```

68.példa

Következő példánk a "számkitalálós játék" (34. példa) újraírása lesz, ezzel a programozási technikával. Hasonlítsuk alaposan össze a két algoritmust.

```
<script type="text/javascript">
var n = Math.floor(Math.random()*1000)+1;  // így globális lesz a változó
function teszt() {
  var tipp = parseInt(document.form1.TIPP.value); // parseInt: Szám típusra alakít!
  var szoveg = '';
  if (tipp > n) { szoveg = 'Kisebb!';}
  if (tipp < n) { szoveg = 'Nagyobb';}
  if (tipp ==n) { szoveg = 'TALÁLT'; }
  document.getElementById('keret1').innerHTML = szoveg
}

</script>
<H3> Gondoltam egy számot 1..1000 között.</H3>
<form name="form1">
     Tipp: <input type="number" name="TIPP" value="0">
     <input type="button" value="OK" onClick="return teszt();">
     <!— A nyomógombra kattintva lefut a "teszt" JS eljárás -->
</form>
<div class="A0" id="keret1"></div>
<!—  ebbe az üres DIV-be írja be a "teszt" függvény az eredményeket (innerHTML)  -->
```

69.példa

Függelék

Vezérlő szerkezetek

```
if (feltétel) {utasítások; }
if (feltétel) {utasítások; } else {utasítások; }

switch(n) {
   case 1:  utasítások;   break;   // ha n = 1
   case 2:  utasítások;   break;   // ha n = 2
   default:  utasítások;           // n != 1 és n!= 2
}
for (változó=kezdő érték; feltétel; ++változó) { utasítások; }  // számláló ciklus

while (feltétel) { utasítások; }                                 // elöl tesztelő ciklus

do { utasítások; }  while (feltétel)                             // hátul tesztelő ciklus


function eljárásnév (paraméterek) {
  utasítások;
return visszatérési_érték; }
```

Operátorok

Értékadó operátorok (x=10; y=5)

Operátor	Művelet	Példa	Eredmény=
+=	x+=y	x=x+y	x=15
-=	x-=y	x=x-y	x=5
***=**	x*=y	x=x*y	x=50
/=	x/=y	x=x/y	x=2
%=	x%=y	x=x%y	x=0

Aritmetikai operátorok (x = 5;)

Operátor	Művelet	Példa	Eredmény
+	Összeadás	x=y+2	x=7
-	Kivonás	x=y-2	x=3
*****	Szorzás	x=y*2	x=10
/	Osztás	x=y/2	x=2.5
%	Modulus (osztás maradéka)	x=y%2	x=1
++	Növelés (+1)	x=++y	x=6
--	Csökkentés	x=--y	x=4

Relációs operátorok (x = 5;)

Operátor	Művelet	Példa
==	egyenlő	x==8 is false
===	egyenlő és azonos típus	x===5 is true
!=	nem egyenlő	x!=8 is true
>	nagyobb	x>8 is false
<	kisebb	x<8 is true
>=	nagyobb, vagy egyenlő	x>=8 is false
<=	kisebb, vagy egyenlő	x<=8 is true

Logikai operátorok (x=6; y=3;)

Operátor	Művelet	Példa
&&	and	(x < 10 && y > 1) is true
||	or	(x==5 || y==5) is false
!	not	!(x==y) is true

Bitművelet operátorok (x=6; y=3;)

Operátor	Művelet	Példa
&	a és b (bitenkénti és)	
|	a vagy b (bitenkénti vagy)	
^	a xor b (bitenkénti xor)	
~	!a (bitengénti negálás: 1. komp.)	
<<	bitenkénti balra tolás	
>>	bitenkénti jobbra tolás	

Globális függvények

Függvény	Művelet	Példa
isFinite(x)	igaz, ha a x szám típus	isFinite("negyven") → false | isFinite(40) → true
isNaN(x)	igaz, ha x nem szám típus	isNaN("negyven") → true | isNan(40) → false
Number(x)	x változót szám típussá alakítja (ha tudja)	document.write(Number("40.5 years")) → 40.5
parseFloat()	x változót valós számmá alakítja (ha tudja)	document.write(parseFloat("40.5 years")) → 40.5
parseInt()	x változót egész számmá alakítja (ha tudja)	document.write(parseInt("40.5 years")) → 40
String()	x változót stringgé alakítja	string(40) → "40"

Beépített objektumok

A Math objektum konstansai (Math.)

Név	Művelet
E	Az Euler szám értékt adja (approx. 2.718)
LN2	Returns the natural logarithm of 2 (approx. 0.693)
LN10	Returns the natural logarithm of 10 (approx. 2.302)
LOG2E	Returns the base-2 logarithm of E (approx. 1.442)
LOG10E	Returns the base-10 logarithm of E (approx. 0.434)
PI	PI értékét adja vissza
SQRT2	Returns the square root of 2 (approx. 1.414)

A Math objektum függvényei (Math.)

abs(x)	Returns the absolute value of x
acos(x)	ARCCOS x értéke radiánban
asin(x)	ARCSIN x értéke radiánban
atan(x)	ARCTAN x értéke radiánban -PI/2 és PI/2 radián között
atan2(y,x)	Returns the arctangent of the quotient of its arguments
ceil(x)	Returns x, rounded upwards to the nearest integer
cos(x)	COS x értéke radiánban
exp(x)	Returns the value of Ex
floor(x)	X egész részét adja eredményül
log(x)	Returns the natural logarithm (base E) of x
max(x,y,z,...,n)	A megadott paraméterek közül a legnagyobbat adja eredményül.
min(x,y,z,...,n)	A megadott paraméterek közül a legkisebbet adja eredményül.
pow(x,y)	Eredménye X az Y-adik hatványon
random()	0..1 intervallumban valós véletlenszámot generál
round(x)	Rounds x to the nearest integer
sin(x)	SIN x értéke radiánban
sqrt(x)	x négyzetgyökét adja eredményül
tan(x)	TAN x értéke radiánban

A Number objektum konstansai

var num = **new** Number(value);

*példa: **var** n = **new** Number(314.15926);*

Név	Művelet
toExponential(x)	Szám alakítása exponenciális formátumra: → 3.1415926e+2
toFixed(x)	Tizedesvessző utáni értékek: toFixed(2) → 314.16
toString()	Számtípus konvertálása stringbe

A String objektum metódusai

var s = **new** String(value); *példa: **var** s = **new** String('Almáspite');*

név	Művelet
length	A string karaktereinek számát adja vissza: **s.length → 9**
charAt(i)	A stringben "i"-edik pozícióján lévő karaktert adja vissza. (Nulladik elemmel kezdődik!): ***s.charAt(2)*** → *"m"*
charCodeAt(i)	Az "i"-edik pozíción lévő karaktert kódját adja vissza**:** ***s.charCodeAt(0) → 65***
concat(s)	S Stringhez fűz stringet**: *s.concat('kusz') → "Almáspitekusz*"**
fromCharCode(i)	ASCII kód karakter értékét (értékeit) adja vissza : ***String.fromCharCode(72,69,76,76,79)*** → *"HELLO"*
indexOf(s)	A paraméterként megadott karakterlánc első előfordulása a stringben: **s.indexOf("pite")** → 5
lastIndexOf(s)	A paraméterként megadott karakterlánc utolsó előfordulása a stringben
substr(i,j)	Karakterlánc kivágása, "i" pozíciótól; "j" hosszan: ***s.substr(5,4)*** → *"pite"*
toLowerCase()	A stringet kisbetűsre konvertálja.
toUpperCase()	A stringet nagybetűsre konvertálja.
valueOf()	A string értékét adja vissza (két string összehasonlításánál fontos!)

Az Array objektum metódusai

var tomb = **new Array**('Uborka','Dinnye','Krumpli','Sajt');

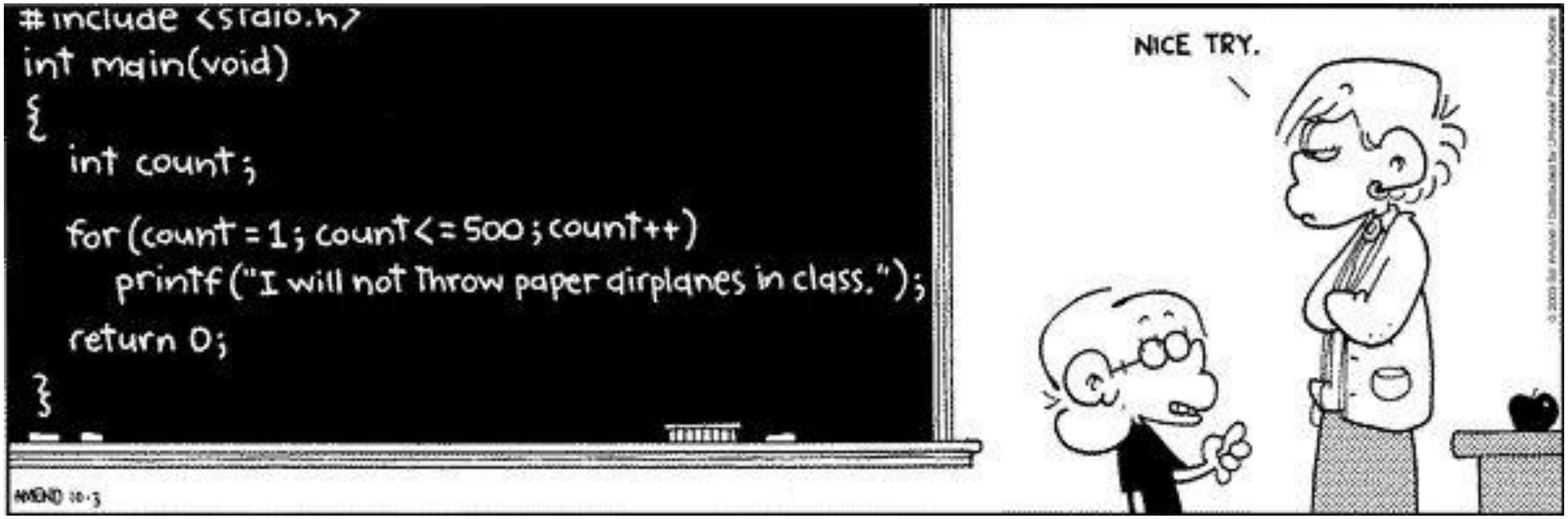
név	művelet
length	A tömb elemszámával tér vissza: **tomb.length → 4**
concat()	Két, vagy több tömböt egyesít
indexOf()	A paraméterként kapott értéket keresi a tömbben, és indexet ad vissza**: tomb.indexOf('Dinnye') → 1**
lastIndexOf()	Ugyanaz mint az "indexof", de az utolsó előfordulást adja vissza.
pop()	Eltávolítja a tömb utolsó elemét**: tomb.pop()** → 'Uborka','Dinnye','Krumpli'
push()	Új elemet fűz a tömb végére: **tomb.push('Sajt')**
shift()	Eltávolítja a tömb első elemét: **tomb.shift()** → 'Dinnye','Krumpli','Sajt'
unshift()	Új elemet szúr be a tömb első poziciójára. **tomb.unshift()**

Források

http://hu.wikipedia.org/wiki/JavaScript : Bevezető

http://www.w3schools.com/jsref/default.asp : JavaScript referencia

http://www.w3schools.com/js/default.asp : JavaScript tutorial


#include <stdio.h>
int main(void)
{
int count;
for (count = 1; count<=500; count++)
printf("I will not Throw paper airplanes in class.");
return 0;
}
NICE TRY.